On-Line Debugging and Performance Monitoring with Barriers

Ernesto Novillo Paul Lu

Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada

E-mail: {ernie|paullu}@cs.ualberta.ca

Abstract

We introduce the Stupid Barrier Tricks *(SBT) library for on-line debugging and performance monitoring of shared-memory parallel programs. Single-program-multiple-data (SPMD) programs often use barriers to synchronize threads of execution and to delimit the start and end of different phases of computation. Through the novel (and simple)* named barriers *construct, dynamic performance warnings, and integration with lightweight performance counter libraries, SBT helps programmers localize deadlocks and performance bottlenecks in their programs.*

SBT is a portable library that currently supports both POSIX threads and SGI Irix `sproc` *threads. SBT also supports both the PCL and Irix* `libperfex` *performance counter libraries. For production runs, the SBT overheads can be eliminated using conditional compilation.*

1. Introduction

Debugging parallel programs is generally acknowledged to be more difficult than debugging sequential programs. In addition to logic errors, parallel programs contain potential synchronization and deadlock bugs. As well, identifying the performance bottlenecks of a parallel program is more complicated than for a sequential program; it is not just a matter of which function consumes most of the execution time, but also the load balance or imbalance between processors.

Of course, there are parallel debuggers and other tools to help the programmer [6, 7]. Debuggers can give a good snapshot view of an executing program, but they do not always give a good trace of the sequence of events leading to an error. Furthermore, there are few widely-available (e.g., open source) cross-platform parallel debuggers with advanced features. Also, some parallel computers are batch scheduled (i.e., job queues), so the interactive use of debuggers becomes impractical or impossible. The infamous `printf()` continues to be popular because it is portable, it provides a transcript of the run-up to an error, and it works for both interactive and batch jobs.

There are performance debugging tools as well (e.g., [11, 14]). However, they are often based on detailed and voluminous trace generation and post-processing, or they can have high run-time overheads. Sometimes, even for moderately-sized runs, the size of the traces becomes too unwieldy. For a high-level view of the performance bottlenecks of a problem, off-line performance debuggers can be too inconvenient to use for many programmers. Even worse, a programmer may not suspect that a particular phase of the computation is a bottleneck and will not investigate further. Of course, for detailed performance debugging, after the bottleneck has been located to a specific phase, these tools continue to be effective and valuable.

Ideally, the programmer should be able to quickly determine where a program is running into errors and dynamically be informed about potential performance problems (i.e., on-line). It is desirable to avoid the extra step of starting up a debugger, starting a trace, or post-processing large amounts of trace data off-line.

We have developed the Stupid Barrier Tricks (SBT) library to support some simple on-line debugging and performance monitoring tasks. For programs written in the single-program-multiple-data (SPMD) style, SBT can be used to locate where programs are hanging, where they are encountering performance bottlenecks, and to gain insight as to why there are bottlenecks. Barriers are synchronization points between phases of computation. No thread can proceed beyond a barrier until all threads have reached the barrier. Consequently, they are natural places to insert lightweight performance collection code that will aid programmers to better understand how the different threads be-have. For production runs, SBT code can be removed via conditional compilation.

Well-known parallel programming systems, including thread libraries (e.g., `sproc` [4]), message-passing libraries (e.g., MPI [12]), and compiler extensions (e.g., OpenMP [5]) include support for barriers.

SBT is designed to help answer the following common questions asked by parallel programmers:

1. Where is my program currently executing? If it is deadlocked, where is it deadlocked?

2. What is the most computationally-intensive phase of my program? Where are the bottlenecks?

3. Is there a load imbalance among threads within a phase? Why is there a load imbalance?

The current version of the SBT library supports programs written in C/C++ with POSIX threads (Pthreads) and SGI Irix's `sproc` threads. SBT can also use the portable Performance Counter Library (PCL) and Irix's `libperfex` to access hardware performance counters. PCL supports Solaris, AIX, Compaq Tru64, and Linux. Consequently, SBT can be ported to a number of different shared-memory platforms. In the future, SBT will be extended to support standard message-passing libraries, such as the Message-Passing Interface (MPI).

2. Stupid Barrier Tricks: An Overview

Stupid Barrier Tricks (SBT) is a library that implements a barrier abstraction and provides on-line debugging and performance information. SBT can aid in the development, debugging, and optimization of a parallel program and then be removed, via conditional compilation, for production runs. Through the use of environment variables, users can control and focus the monitoring efforts of SBT without recompiling their code.

Parallel programs are often divided into phases that are delimited by barriers. Single-program-multiple-data (SPMD) style programs, in particular, use barriers for synchronization (Figure 1). Master-worker and client-server style programs are not as likely to use barrier synchronization, therefore SBT may not be suitable for those programs.

Normally, barriers are anonymous, like forks and joins, but SBT introduces the simple concept of a *named barrier* as a way to produce a low-noise trace of the progress of a parallel program. In our experience, programmers often use `printf()`s to accomplish the same task. A named barrier at the beginning of, say, the initialization phase produces the output "`Start Initialization`"; a different named barrier at the end of the phase produces the output "`End Initialization`" (Figure 1). Therefore, by watching

```
void main( int argc, char ** argv )
{
  BARRIER;         /* Anonymous barrier */
  ...
                   /* Named barrier */
  N_BARRIER( "Start Initialization" );

  ... /* Code for initialization phase */

                   /* Named barrier */
  N_BARRIER( "End Initialization" );
  ...
}
```

Figure 1. Anonymous and Named Barriers in SPMD Programs

the standard output of the program, the programmer can see where the program is currently executing. If the end-of-phase message is not seen, the programmer knows that either a deadlock has occurred, or the phase of computation is long. Also, any user-defined output during the phase is bracketed by the output of the named barriers. Thus, named barriers label and associate output with the corresponding phase of the program.

Both anonymous barriers (which can still be identified by their line number in the source code file) and named barriers provide natural caliper and watchpoints for performance monitoring. After compiling their parallel program to use SBT, users set an environment variable to identify the barrier they wish to watch (either using its name or its line number). While the program is executing, the following information will be dynamically collected and selectively generated:

1. *Phase time*: The amount of wall-clock time spent between the barrier at the beginning of a phase and the barrier at the end.

   All barriers, either implicitly or explicitly, represent the end of one phase and the start of another phase. In this way, the most computationally-intensive phases are easily identified.

2. *Barrier time*: The amount of time spent by the program at a barrier.

   By definition, barrier time is the time difference between when the first thread arrives at the barrier and when the last thread arrives. Long barrier times suggest that performance is being lost due to idle threads at the barrier. Poor load balancing is a common cause of long barrier times.

3. *Inter-thread arrival time*: The time difference between one thread's arrival and the next thread's arrival.

   The order in which threads arrive at the barrier is also noted. When locating load balancing problems, it can be important to know the order of and interval between thread arrivals.

4. *Performance metrics and hardware counters*: Depending on the CPU architecture, information about cache misses, graduated instructions, CPU cycles, floating-point operations, etc., can be collected by hardware counters.

   Low-level performance counters can give insight as to what might be the cause of a performance problem. Poor memory locality, poor load balancing, and high synchronization rates (i.e., poor granularity) can be revealed by examining performance counters.

Additionally, warnings are automatically issued for barriers that are particularly costly (e.g., barrier times longer than a user-selectable threshold). The relevance and frequency of these warnings can also be parametrized by user-controlled environment variables.

Now, we consider two illustrative examples of how SBT is used in practice. The matrix multiplication and LU-decomposition examples that follow are not tuned for high performance; they are simple examples to demonstrate the features of SBT.

2.1. Example: Matrix Multiplication

The matrix multiplication example uses SBT barriers to synchronize threads at two points of the execution: one after initializing the matrices to be multiplied, and the other at the end of the multiplication itself (lines 18 and 22 of Figure 2). Work and data are block-distributed between threads. According to this work distribution, each thread is responsible of multiplying a range of contiguous rows of matrix A by matrix B. The result of the multiplication is stored in C.

We use SBT with Pthreads and the PCL library for this example. Therefore, the code is linked to 3 libraries: `libpthread`, `libsbt`, and `libpcl`. For simplicity, the number of threads to launch and the matrix size are fixed according to the definitions of `THREAD_COUNT` and `SIZE` in lines 6 and 7 of the code, respectively. In general, SBT (and the other libraries) can support any number of threads and problem size.

There are two defines and one include that have to be provided *before* including the SBT header file, `sbt_barrier.h` (lines 1, 2, and 3 of Figure 2):

1. `SBT_BARRIER`: A macro that identifies the specific barrier data structure (of type `sbt_barrier_t`) to be used implicitly for both anonymous and named barriers.

   Structure `sbt_barrier_t` contains a pointer to either an `sproc` barrier or to our implementation of a Pthreads-based barrier, depending on which threading library is specified. Pthreads does not provide a barrier function, so we have implemented one.

   The `sbt_barrier_t` data structure also contains various fields used for gathering performance statistics.

2. `SBT_THREADID`: A macro that identifies a function or variable that provides the numerical identity (between 0 and $n-1$ for n threads) of the current thread. It is used by `BARRIER` and `N_BARRIER()` when outputting thread-specific information.

```
 1  #define SBT_BARRIER     my_barrier
 2  #define SBT_THREADID    my_id
 3  #include <pthread.h>
 4  #include "sbt_barrier.h"
 5
 6  #define THREAD_COUNT    4
 7  #define SIZE            512
 8
 9  SBT_BARRIER_T SBT_BARRIER;
10  int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
11
12  void* work( void* id )  {
13      int my_id = *(int*)id;
14      int row_count = SIZE/THREAD_COUNT;
15      int first_row = (int)my_id*row_count;
16      /* initialize my block of A and B */
17      init_block( first_row, row_count );
18      N_BARRIER( "End initialization" );
19      /* multiply row_count rows of A by B,
20         starting in first_row */
21      matrix_multiply();
22      N_BARRIER( "End multiplication" );
23      return( NULL );
24  }
25
26  int main()  {
27      pthread_t threads[THREAD_COUNT];
28      int thread_ids[THREAD_COUNT], i;
29      /* initialize library and barrier */
30      SBT_BARRIER = sbt_init( THREAD_COUNT );
31      /* launch threads */
32      for( i=1; i<THREAD_COUNT; i++ )  {
33          thread_ids[i] = i;
34          if( pthread_create( &(threads[i]),
35              NULL, work,
36              (void*)&(thread_ids[i]) ) ) {
37              printf( "error creating "
38                      "thread %d\n", i );
39              return( 1 );
40          }
41      }
42      thread_ids[0] = 0;
43      work( (void*)&(thread_ids[0]) );
44      /* free library resources */
45      sbt_finalize();
46      return( 0 );
47  }
```

Figure 2. Matrix multiplication: SBT barriers for Pthreads.

```
 1  SBT options (built for pthreads)
 2          SBT_WATCH       End multiplication
 3          SBT_WATCH_ALL   0
 4          SBT_WARNINGS    1
 5          SBT_WARN_TIME   1000
 6          SBT_PHASE_TIMES 0
 7          SBT_NO_DEBUG    0
 8          SBT_EVENTS      yes
 9          SBT_PCLTHREADID 1
10                  PCL event 0:    PCL_L2CACHE_MISS
11                  PCL event 1:    PCL_INSTR
12
13  SBT "End initialization"
14          (barrier: 0ms, phase 1: 19.923s,
15           from init: 19.924s)
16
17  SBT barrier watch in pmm.c:49 "End multiplication"
18          Barrier time:   5 ms
19          Phase time:     6.690 sec
20          Total time:     26.614 sec
21          Order of arrival:
22        inter      from                real
23   id  thread      init                time
24    0     0ms     26.608s         16:55:16.936
25    1     5ms     26.614s         16:55:16.942
26
27   SBT: Phase 2 PCL counter information
28    id        L2CACHE_MISS               INSTR
29     1             8302132          1747606057
30
31
32   SBT: Overall accumulated PCL counter values
33    id        L2CACHE_MISS               INSTR
34     1            10975177          2005908351
```

Figure 3. Example SBT output for matrix multiplication: 512×512 matrices, 2 Pthreads, PCL, Linux.

Env. Variable	*Value*
SBT_WATCH	End multiplication
SBT_EVENTS	PCL_L2CACHE_MISS:PCL_INSTR
SBT_WATCHTHREAD	1

Table 1. SBT environment variables in matrix multiplication.

3. Threading library: The header file for the appropriate threading library.

   Currently, either `#include <pthread.h>` for Pthreads or `#include <sys/prctl.h>` for `sproc` threads are supported.

In this example, threads use two named barriers: one after initializing their own block of the matrices (i.e., line 18, `End initialization`) and another one after their part of the computation is completed (i.e., line 22, `End multiplication`).

The output in Figure 3 corresponds to a multiplication of two integer matrices of size 512×512 with 2 threads under Linux. Hypothetically, if there had been a logical bug in the program and a deadlock occurred within function `matrix_multiply()`, the user would have seen the output `End initialization` and then no more output. Since the programmer knows there is a named barrier called `End multiplication` that comes after the first named barrier, the immediate suspicion is that the program is hung somewhere between the two named barriers. If necessary, named barriers can be easily added to and removed from a program to further triangulate the problem. Without named barriers, the programmer would either have to attach to the threads using a debugger or add `printf()`s to discover where the program is stopped. Therefore, named barriers provide a convenient way to localize the problem quickly and easily. For the remainder of the discussion, we ignore the possibility of deadlock.

Table 1 shows the environment variables that are used, along with their values. For example, under the C shell, the user enters `setenv SBT_WATCH "End multiplication"` before running the matrix multiplication. The first of the three variables, `SBT_WATCH`, directs SBT to watch the named barrier `End multiplication`. Therefore, performance monitoring data is displayed when this barrier is reached. Barrier time, phase time, total time from library initialization (i.e., when `sbt_init()` is called), and inter-thread arrival times are given (lines 17 to 25, Figure 3). The `End initialization` named barrier produces fewer lines of output (lines 13 to 15) because it is named, but it is not watched. Anonymous barriers do not normally produce any output.

Second, a colon-separated list of PCL events to be monitored is specified by environment variable `SBT_EVENTS` (Table 1). PCL is a portable interface to hardware counters for events such as Level 2 cache misses and the number of instructions executed by the CPU. The performance counter information is output for every phase that ends at a watched barrier. Cumulative totals for all phases are shown at the end of the computation (lines 32 to 34, Figure 3) when `sbt_finalize()` is called (line 45, Figure 2).

Third, the `SBT_WATCHTHREAD` environment variable is set to 1, so thread number 1 will be the one to start and stop the counters, as well as store and accumulate their values, at the end of every phase. Currently, PCL is not thread-safe, so SBT can only gather performance counter information for one thread. Multiple runs would be required to gather performance numbers for all threads. The Irix `sproc` and `libperfex` version of SBT supports per-thread hardware counters within a single run (Section 2.2).

2.2. Example: LU-Decomposition

We now consider two different versions of a naive implementation of LU-decomposition. Our purpose is to illustrate how SBT is used in practice. The first version uses block data distribution. The second version uses cyclic data

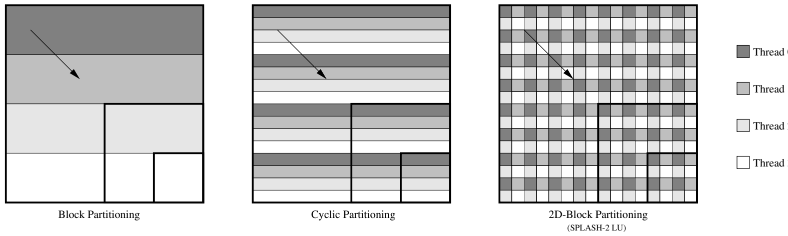

Figure 4. LU-decomposition work partitioning strategies.

Number of Processors	*Speedups*	
	Block dist.	*Cyclic dist.*
2	2.62	2.68
4	4.14	5.56
8	8.09	9.32

Table 2. LU-decomposition speedups: 2048×2048 matrix and different distributions.

distribution. We have also successfully applied SBT barriers to the highly-tuned SPLASH-2 implementation of LU-decomposition [13] to gain experience in porting existing code to SBT.

Figure 4 gives an intuitive idea of the different work partitioning strategies assuming 4 threads. With block distribution, each thread is responsible for reducing a contiguous set of rows, while with cyclic distribution each thread reduces rows in round-robin fashion. Finally, 2D-block partitioning divides the matrix in blocks along both axes and assigns blocks to processes in an interleaved fashion.

An in-depth analysis of the differences in performance between the implementations represents a good example of how SBT can produce useful and insightful information of an application's run-time behavior. As with any set of similar parallel algorithms, real time and speedups are the first criteria for comparison. We ran the two programs on an SGI Origin 2000, using `sproc` threads, decomposing a 2048 × 2048 matrix on 2, 4, and 8 processors (Table 2). The speedups are calculated comparing the parallel versions against a purely sequential algorithm. The super-linear speedups are due to well-known cache effects. We are only interested in the differences between block and cyclic distribution, instead of absolute numbers.

The empirical speedups indicate that cyclic work partitioning achieves higher performance than block work partitioning, which is a well-known property of LU-decomposition. Again, we are using LU-decomposition merely to demonstrate the capabilities of SBT.

However, suppose the programmer first implemented LU-decomposition using block distribution (block-LU, for short). While running block-LU, output like that shown in Figure 5 would be generated. The figure is a subset of the entire output and corresponds to the 1,071'th loop iteration of the LU-decomposition. The warning in line 28 is generated because the barrier time exceeded the threshold of 100 ms specified by environment variable `SBT_WARN_TIME` (Table 3).

The output tells the programmer that thread 0 arrived at the barrier first (line 8, Figure 5) and thread 4 arrived last (line 15). Some of the threads arrived at nearly the same time, but there is an inter-thread arrival gap of 49 ms, 6 ms, and 55 ms between some threads, which explains why the barrier time is a relatively large 112 ms. But it is still unclear as to why there are large gaps in the inter-thread arrival times.

To provide more insight, hardware performance counters can be used. In particular, a value of `0` for `libperfex` environment variable `T5_EVENT0` selects thread-by-thread CPU cycle counts for monitoring. Therefore, we can see that the 8 different threads of block-LU execute between 3,579 and 10,832,578 cycles in the LU-decomposition loop before hitting the barrier (lines 17 to 26, Figure 5). Thread 0 is arriving at the barrier first because it does little work in loop iteration 1,071 and it sits idle at the barrier waiting for the other threads after only 3,579 CPU cycles. In contrast, thread 7 executes for 10,832,578 CPU cycles before reaching the barrier. Due to normal non-determinism in the system, the threads may arrive at the barrier in a different order than the CPU cycle counts would indicate (i.e., thread 4 arrives after thread 7). However, the CPU cycle counts do provide a rough partial order of expected thread arrival orders.

Together, the thread inter-arrival times and performance counters indicate that there is a load balancing problem. As can be seen from Figure 4, by loop iteration 1,071 of 2,048, half of the threads have no more work to do. Therefore, block-LU achieves relatively low speedups (compared to cyclic distribution) due to a flaw in its work partitioning strategy.

In contrast, the cyclic work distribution scheme proves

Environment Variable	*Value*
SBT_WATCH_ALL	1
SBT_WARN_TIME	100
T5_EVENT0	0 (CPU cycles)

Table 3. SBT environment variables used with LU-decomposition.

```
 1  SBT barrier watch in par-lu.c:320
 2          Barrier time:   112 ms
 3          Phase time:     0.113 sec
 4          Total time:     517.114 sec
 5          Order of arrival:
 6        inter      from                real
 7   id  thread      init                time
 8    0     0ms     517.002s        19:24:49.589
 9    1     0ms     517.002s        19:24:49.589
10    2     0ms     517.002s        19:24:49.589
11    3     0ms     517.002s        19:24:49.589
12    5    49ms     517.052s        19:24:49.639
13    6     0ms     517.052s        19:24:49.639
14    7     6ms     517.059s        19:24:49.646
15    4    55ms     517.114s        19:24:49.701
16
17  SBT: Phase 1071 perfex counters
18   id     Event 0 Count
19    0        3579
20    1        4005
21    2        4165
22    3        3984
23    4     8970877
24    5     9570578
25    6     9640210
26    7    10832578
27
28  SBT WARNING: barrier in par-lu.c:320
29          (barrier: 112 ms > 100 ms,
30           phase 1071: 0.113, from init: 517.114s)
31
32  SBT Phase 1071 time: 0.113 sec
```

Figure 5. Example SBT output for LU-decomposition: Block distribution, 8 `sproc` threads.

```
 1  SBT barrier watch in par-lu.c:287
 2          Barrier time:   10 ms
 3          Phase time:     0.051 sec
 4          Total time:     621.791 sec
 5
 6  ...inter-thread arrival times omitted...
 7
 8  SBT: Phase 1071 perfex counters
 9   id     Event 0 Count
10    0     8222699
11    1     7743750
12    2     7819229
13    3     8055816
14    4     7609739
15    5     7961336
16    6     7767987
17    7     8125033
18
19  SBT Phase 1071 time: 0.051 sec
```

Figure 6. Example SBT output for LU-decomposition: Cyclic distribution, 8 `sproc` threads.

to be much better balanced, as can be seen in Figure 6. The distribution of work among different processes is roughly equal (lines 8 to 17, Figure 6), with all processes doing approximately the same amount of work throughout all of the iterations.

If the programmer had not previously been aware of the benefits of cyclic work distribution for LU-decomposition, the first warning that there might have been a performance problem would have been a barrier time warning (similar to line 28, Figure 5). By watching the indicated barrier and using hardware performance counters, the programmer learns that the cause of the warning is a load imbalance between the threads. Then, the programmer can fix the performance bottleneck by implementing, for example, a cyclic work distribution strategy.

Finally, we ported SPLASH-2's LU-decomposition code to use Pthreads, `sproc`, and the SBT barriers. A set of null macros provided with SPLASH-2 was implemented for both threading models to provide parallelism and synchronization. The SPLASH-2 LU code itself underwent little change. Only the required SBT `#include` and `#define`'s (as explained in Section 2.1), along with calls to `sbt_init()` and `sbt_finalize()`, were added. The existing calls to the `BARRIER` macro needed no change. Although the code was originally optimized to use 2D-block work distribution, we introduced a quick modification to support any of the three work distribution techniques shown in Figure 4. Preliminary test runs with the ported code confirm that block work distribution is the least efficient of the three due to poor load-balancing, while the better balanced work distribution techniques, cyclic and 2D-block, produce faster decompositions.

3. Implementation

Currently, SBT supports two different threading libraries: POSIX threads (Pthreads) and SGI Irix's `sproc` threads. Given that hardware event counting is an architecture-specific capability, it is not trivial to implement this functionality for a wide array of microprocessors. Thus, hardware counting capability is provided in two different flavors: the Performance Counter Library (PCL) and Irix's `libperfex`. PCL is portable and supported on several different architectures, including Solaris, AIX, Compaq Tru64, and Linux. Pthreads is also available on these platforms. By layering SBT on top of Pthreads and PCL, SBT is a relatively portable library. We are also considering supporting the PAPI [2] interface to hardware counters.

SBT supports `sproc` and `libperfex` because those Irix capabilities have some advanced features not found elsewhere, including per-thread performance counter capabilities [3].

For all versions of SBT, it is possible to remove the overheads of performance monitoring by defining the compile-time label `SBT_OFF` and recompiling (e.g., `-DSBT_OFF` on the compile line). This version of the library contains just the bare calls to the barriers and has no additional overhead compared to calling the barrier synchronization directly from the program. Therefore, production runs of programs using SBT do not incur unnecessary overheads.

The first building block of SBT for Pthreads is a simple barrier primitive that makes use of a mutex and a condition variable. There is no barrier function in Pthreads proper. As threads arrive at the barrier, they acquire the mutex and enter a critical section to increment a counter and suspend themselves on the condition variable (this automatically relinquishes the mutex). The last thread to arrive restarts its siblings by broadcasting a signal for the condition, and resets the counter to 0, leaving it ready for the next barrier call. It is possible to use a higher-performance barrier implementation, if desired.

SBT uses a variety of shell environment variables (too numerous to detail here) to specify the watched barriers, thresholds for performance warnings, and the verbosity of SBT's output. The values of the SBT environment variables are read and interpreted during library initialization (i.e., `sbt_init()`, Figure 2). These values are kept in a single instance of data structure `sbt_env_t` that defines the behavior of all the barriers in a program. SBT's initialization function returns a fully functional instance of `sbt_barrier_t` that is allocated and initialized internally as one of the fields in the library's `sbt_env_t` variable. When the library is compiled with the `SBT_OFF` flag, the `sbt_env_t` variable becomes just a placeholder for the instance of `sbt_barrier_t` that is returned upon initialization.

An anonymous SBT barrier can be inserted with the `BARRIER` macro; a named barrier uses macro `N_BARRIER("barrier name")`. The difference between anonymous and named barriers is simply that when information is output, anonymous barriers will be referred to with the pair `file_name:line_number`, while named barriers will output their assigned names after the file and line number. Additionally, barrier time, phase time, and time from initialization will be output by default for named barriers.

As threads enter the barrier, they follow a path similar to the barrier primitive described earlier: enter critical section, increment a counter, relinquish mutex. Some of the performance information, however, needs to be gathered with mutual exclusion, so a timestamp and a thread number are stored before leaving the critical section. After having stored this information and exited the critical section, threads call the actual underlying barrier. Following the synchronization point and the information-gathering code, one of the threads is assigned the task of outputting the required information for the current barrier (e.g., barrier time, phase time, hardware counters, warnings, etc).

After all threads have terminated their assigned tasks, a call to `sbt_finalize()` will print out hardware counter accumulated totals as well as free all the dynamically allocated memory for the barrier.

The inner workings of SBT for `sproc` threads are similar to Pthreads. There are, of course, differences in how threads are created and the required data structures to support shared memory under Irix [4]. Synchronization in this threading model is achieved through the use of an *arena*, a region of shared memory to which all processes of an application have to attach. Irix provides library functions for locks, semaphores, and (notably) barriers, the first and latter of which are used by SBT.

3.1. Hardware Performance Counters

Contemporary CPU architectures support hardware counters for important low-level events that affect performance. Data such as cache miss counts, number of CPU cycles, number of graduated instructions, and floating point operations completed, can be crucial to the process of finding a program's bottlenecks. Since the counters are implemented in hardware, they also incur fewer overheads than software-instrumented programs.

The main concern regarding the implementation of hardware counting for SBT is portability. The libraries to access the counters and the specific counters that are available on different CPU architectures differ from system to system. The Performance Counter Library (PCL) [1] is a useful tool since it provides a unified API to the hardware counters on a number of different architectures. One of the drawbacks of PCL is that, as of version 1.3, is not thread safe. Nevertheless, SBT can be linked to PCL, and although the hardware counter information can only be gathered for one thread, the user can specify and vary the thread to watch.

As a means of getting counting information for individual processors, SBT can be alternately linked to `libperfex` [3], SGI's performance counting library. This functionality was developed and tested on an SGI Origin 2000 with MIPS R10000 processors. We are also considering supporting the PAPI [2] interface to hardware counters.

For both PCL and `libperfex`, hardware counting is started and stopped at the beginning and end of each phase, respectively. This translates into a function call to stop the counters at the beginning of the barrier code, and another call to restart counting before returning. As the computation progresses, the counter values for each phase are output (individually for each thread in the case of `libperfex`, and only for the specified thread in PCL), and then accumulated and checked for overflows. At the end of the computation the accumulated values are shown.

4. Related Work

SBT is a re-implementation and significant extension of the named barriers in the Aurora Distributed Shared Data system [9, 10]. Many other parallel programming systems, debuggers, and performance analysis tools have incorporated similar concepts [7]. Named barriers are most similar to the *named parallel regions* of the commercial GuideView tool for OpenMP [8]. However, SBT is unique in its emphasis on using pairs of barrier synchronization points to delimit the phases of particular interest (e.g., dynamic framework for metrics), on-line performance monitoring and warnings (e.g., no trace generation or post-processing), and its implementation strategy (e.g., open-source library).

Named barriers are also similar to breakpoints and watchpoints in parallel debuggers, such as TotalView [6]; all are techniques to identify points of interest in a program. Most debuggers and performance monitoring tools provide metrics on a function body, basic block, or line number basis (e.g., [6, 11, 14]). Especially in SPMD programs, barriers are more natural delimiters of the program phases.

As with lock metering utilities [4], SBT addresses a specific class of programs and a subset of all possible correctness and performance problems. Other tools and systems can potentially address a larger set of problems. Of course, as an application library, SBT may be used in combination with debuggers and other performance analysis tools.

5. Concluding Remarks

SBT is a portable library that helps programmers locate where their shared-memory programs are spending their time (or hung due to a deadlock) and provides insight as to why there may be performance problems. Using the novel named barrier construct, a SPMD parallel program generates a low-noise trace of its execution. The beginning and end of each computational phase is conveniently labelled and automatically measured. When deadlocks occur, it is clear in which phase the error is located. When bottlenecks occur, the programmer can watch the relevant barrier to gather more performance information.

Important metrics, such as phase time, barrier time, and inter-thread arrival time at a barrier are automatically gathered by SBT. If the metrics are outside of an expected range, warnings are generated. The programmer can also access hardware performance counters to gain more insight into load balancing and data locality problems. For production runs, the overheads incurred by SBT can be removed through conditional compilation.

In the future, we plan on extending SBT to support message-passing interfaces, such as MPI. We have also begun work on *loop barriers*, for barriers inside loops. To reduce the verbosity of the output, loop barriers accumulate performance metrics instead of providing per-iteration output.

The current version of SBT can be found at: `http://www.cs.ualberta.ca/~paullu/SBT/`

References

[1] R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 1.3). Technical report, Central Institute for Applied Mathematics, Research Centre Juelich GmbH, 1999.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[3] D. Cortesi et al. *Origin2000 (TM) and Onyx2 (TM) Performance Tuning and Optimization Guide*. Silicon Graphics, Inc., 1998.

[4] D. Cortesi et al. *Topics in IRIX Programming*. Silicon Graphics, Inc., 2000.

[5] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Jan./Mar. 1998.

[6] Etnus LLC. TotalView Multiprocess Debugger/Analyzer. `http://www.etnus.com/Products/TotalView/`.

[7] M. Gerndt, B. Mohr, and B. Miller. Performance Analysis and Tuning of Parallel Programs: Resources and Tools. In *Tutorial at Supercomputing 2000*, Dallas, Texas, November 2000.

[8] Kuck & Associates, Inc., an Intel Company. GuideView. `http://www.kai.com/parallel/kappro/guideview/`.

[9] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 467–473, Geneva, Switzerland, April 1997. Available at `http://www.cs.ualberta.ca/~paullu/`.

[10] P. Lu. *Scoped Behaviour for Optimized Distributed Data Sharing*. PhD thesis, University of Toronto, Toronto, ON, Canada, January 2000.

[11] B. Mohr, A. D. Malony, and J. E. Cuny. TAU. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[12] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[14] Z. Xu, J. R. Larus, and B. P. Miller. Shared-Memory Performance Profiling. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP-97)*, volume 32, 7 of *ACM SIGPLAN Notices*, pages 240–251, New York, June 18–21 1997. ACM Press.